สุทธิศักดิ์ พงศ์ธนาพาณิช

เปิดโลกการเข้ารหัสข้อมูลด้วยภาษาซี ตอน Arithmetic coding (ตอนจบ)

เทคนิคการบีบอัดข้อมูลด้วยวิธี Arithmetic coding ด้วยซอร์สโค้ดของโปรแกรมตัวอย่าง

จากตอนที่ผ่านมาผมได้อธิบายถึงหลักการและอัลกอริทึมของการเข้ารหัสข้อมูลแบบ Arithmetic coding และขั้นตอนการทำงานของอัลกอริทึมการบีบอัดและขยายขนาดข้อมูล ซึ่งรายละเอียดทั้งหมดในตอนที่ผ่านมาผู้อ่านก็คงจะสามารถสร้างเป็นโปรแกรมได้แล้วในบางส่วน ดังนั้นในบทความตอนนี้ผู้เขียนจึงขอจบเทคนิคการบีบอัดข้อมูลด้วยวิธี Arithmetic coding ด้วยซอร์สโค้ดของโปรแกรมตัวอย่างเอาไว้แต่เพียงเท่านี้ และสำหรับในตอนต่อไปก็จะมาว่ากันต่อในเรื่องของการปรับปรุงประสิทธิภาพโมเดลของการบีบอัดข้อมูล ทั้งนี้ก็เพื่อให้ผลลัพธ์ของการบีบอัดข้อมูลด้วยเทคนิค Arithmetic coding ดีขึ้น เพราะจากรายละเอียดทั้งหมดที่ผู้เขียนได้อธิบายถึงการบีบอัดข้อมูลด้วยเทคนิคแบบ Huffman coding หรือ Arithmetic coding ก็ตาม ล้วนแต่มีโมเดลในรูปแบบของสายสตริงเท่านั้น ซึ่งก็หมายความว่าในการบีบอัดข้อมูลแต่ละครั้ง อัลกอริทึมการบีบอัดก็จะต้องทำการสร้างตารางความถี่ของข้อมูลแต่ละตัว และอัลกอริทึมการบีบอัดจะมีประสิทธิผลก็ต่อเมื่อมันสามารถที่จะทำนายตารางความถี่ของข้อมูลได้อย่างถูกต้องแม่นยำ และเมื่อสามารถสร้างตารางความถี่ได้แล้วก็จัดการแจกแจงจำนวนบิตให้กับข้อมูลแต่ละตัว โดยความยาวของจำนวนบิตของข้อมูลแต่ละตัวก็จะแปรผกผันหรือตรงข้ามกับจำนวนความถี่ของมันเอง ดังนั้นผู้อ่านก็จะเห็นว่าถ้าหากเราปรับปรุงโมเดลโดยให้เทคนิคต่าง ๆ สามารถใช้โมเดลที่สูงขึ้นในการบีบอัดข้อมูล ก็จะทำให้ผลลัพธ์ที่ได้ด้วยขึ้นด้วยเช่นกัน (แต่อาจจะมีผลอย่างมากต่อความเร็วในการประมวลผลของโปรแกรม) ซึ่งเราจะมาว่ากันอย่างละเอียดอีกครั้งในบทความตอนถัดไปครับ

จากที่ผู้เขียนได้นำเสนอบทความด้านการบีบอัดข้อมูลด้วยภาษาซีมาหลายตอน ก็มีผู้อ่านหลายท่านได้เขียนจดหมายมายังตู้ป.ณ. 35 ปทฝ.ดินแดง เพื่อสอบถามปัญหากันพอสมควร และก็มีผู้อ่านท่านหนึ่งจากจังหวัดนครปฐมได้สอบถามถึงแหล่งข้อมูลสำหรับการค้นคว้าเพิ่มเติมด้านนี้ ซึ่งผู้เขียนเห็นว่าเป็นประโยชน์อย่างยิ่งต่อผู้อ่านที่สนใจ จึงขอนำมาลงในบทความตอนนี้ สำหรับแหล่งค้นคว้าของผู้เขียนก็ได้จากหนังสือและบทความต่างประเทศต่อไปนี้

- "LZW Data Compression", Dr.Dobb's Journal, Vol.14, Num.10, Oct'89 By Mark Nelson
- "Data Compression", Computer Science Press, 1988 By J.A.Storer
- "Technique of High-Perormance Data Compression", IEEE Computer, Vol.17, Num.6, Jun'84 By Terry Welch
- "Data Compression", John Wiley & Sons, 1989 By Held Gilbert
ซอร์สโค้ดของโปรแกรม LHARC.EXE by Yoshi

ผู้อ่านท่านใดที่สนใจก็เลือกแหล่งค้นคว้าเพิ่มเติมได้ตามสะดวกครับ แต่ถ้าหากไม่ต้องการอ่านบทความหรือเนื้อหาที่เป็นภาษาอังกฤษให้เสียเวลา ก็ให้ติดตามจากบทความนี้ของผู้เขียนก็แล้วกัน เพราะเนื้อหาส่วนใหญ่ที่เป็นภาคทฤษฎีผู้เขียนก็ได้จากการรวบรวมเอาจากหนังสือและวารสารต่างประเทศ แล้วนำมาเรียบเรียงเพื่อให้ง่ายต่อการเรียนรู้และจะเรียงลำดับเนื้อหาจากเทคนิคพื้นฐานง่าย ๆ ไปสู่เทคนิคขั้นสูงที่มีความยากขึ้นตามลำดับ

สำหรับซอร์สโค้ดของโปรแกรมบีบอัดข้อมูลด้วยเทคนิค Arithmetic coding ก็คล้ายกับซอร์สโค้ดที่ผ่านมา โดยผู้เขียนได้รวบรวมเอารูทีนทั้งหมดสำหรับการบีบอัดข้อมูลด้วยเทคนิค Arithmetic coding เอาไว้ในไฟล์ชื่อ ARITH.C ส่วนไฟล์ ARITMAIN.C ก็จะเป็นไฟล์ไดรเวอร์สำหรับเรียกรูทีนการบีบอัดข้อมูล CompressFile หรือการขยายขนาดข้อมูล ExpandFile นั่นเอง สำหรับบทความตอนนี้ก็ขอจบเพียงเท่านี้ พบกันใหม่ฉบับหน้าครับ

ลิสติงที่ 1
ARITMAIN.H

```
/*
 * ARITMAIN.H
 */

#ifndef _ARITH_H
#define _ARITH_H

void CompressFile();
void ExpandFile();

extern char *Usage;
extern char *CompressionName;

#endif  /* _ARITMAIN_H */
```

ลิสติงที่ 2
ARITMAIN.C

```
/*
 * ARITMAIN.C After this is built into a compression program of any sort,
 *            the program can be called like this:
 *
 *          arith infile outfile option
 *          infile          Input filename.
 *          outfile         Output filename.
 *          option -c       Compress data file.
 *                 -e       Expand data file.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bitio.h"
#include "aritmain.h"

void usage_exit();
void print_ratios();
long file_size();

int main( argc, argv )
int argc;
char *argv[];
{
      setbuf( stdout, NULL );
      if ( argc < 3 )
            usage_exit( argv[ 0 ] );

      if ( strcmp( argv[ 3 ], "-c" ) == 0 ) {
      BIT_FILE *output;
      FILE *input;

      input = fopen( argv[ 1 ], "rb" );
      if ( input == NULL )
            fatal_error( "Error opening %s for input\n", argv[ 1 ] );
      output = OpenOutputBitFile( argv[ 2 ] );
      if ( output == NULL )
      fatal_error( "Error opening %s for output\n", argv[ 2 ] );
      printf( "\nCompressing %s to %s\n", argv[ 1 ], argv[ 2 ] );
      printf( "Using %s\n", CompressionName );
      CompressFile( input, output, argc, argv );
      CloseOutputBitFile( output );
      fclose( input );
      print_ratios( argv[ 1 ], argv[ 2 ] );
      return( 0 );
    } else if ( strcmp( argv[ 3 ], "-e" ) == 0 ) {
      FILE *output;
      BIT_FILE *input;

      input = OpenInputBitFile( argv[ 1 ] );
      if ( input == NULL )
      fatal_error( "Error opening %s for input\n", argv[ 1 ] );
      output = fopen( argv[ 2 ], "wb" );
      if ( output == NULL )
      fatal_error( "Error opening %s for output\n", argv[ 2 ] );
```

→

ลิสติงที่ 2 (ต่อ)

```
    printf( "\nExpanding %s to %s\n", argv[ 1 ],
    argv[ 2 ] );
    printf( "Using %s\n", CompressionName );
    ExpandFile( input, output, argc, argv );
    CloseInputBitFile( input );
    fclose( output );
    putc( '\n', stdout );
    return( 0 );
  } else
    usage_exit( argv[ 0 ] );
}

void usage_exit( prog_name )
char *prog_name;
{
    char *short_name;
    char *extension;

    short_name = strrchr( prog_name, '\\' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, '/' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, ':' );
    if ( short_name != NULL )
        short_name++;
    else
        short_name = prog_name;
    extension = strrchr( short_name, '.' );
    if ( extension != NULL )
        *extension = '\0';
    printf( "\nUsage: %s %s\n", short_name, Usage );
    exit( 0 );
}

#ifndef SEEK_END
#define SEEK_END 2
#endif
```

→

```
long file_size( name )
char *name;
{
    long eof_ftell;
    FILE *file;

    file = fopen( name, "r" );
    if ( file == NULL )
        return( 0L );
    fseek( file, 0L, SEEK_END );
    eof_ftell = ftell( file );
    fclose( file );
    return( eof_ftell );
}

void print_ratios( input, output )
char *input;
char *output;
{
    long input_size;
    long output_size;
    int ratio;

    input_size = file_size( input );
    if ( input_size == 0 )
        input_size = 1;
    output_size = file_size( output );
    ratio = 100-(int)( output_size * 100L / input_size );
    printf( "\nInput bytes: %ld\n", input_size );
    printf( "Output bytes: %ld\n", output_size );
    if ( output_size == 0 )
        output_size = 1;
    printf( "Compression ratio: %d%%\n", ratio );
}
```

■

ลิสติงที่ 3 ARITH.C

```
/*
 * This is the arithmetic coding module. Compile with BITIO.C, ARITMAIN.C.
 */
#include <stdio.h>
#include <stdlib.h>
#include "bitio.h"

/*
 * The SYMBOL structure is what is used to define a symbol in arithmetic coding terms.
 * A symbol is defined as a range between 0 and 1. Since we are using integer
 * math, instead of using 0 and 1 as our end points, we have an integer scale.
 * The low_count and high_count define where the symbol falls in the range.
 */

typedef struct {
    unsigned short int low_count;
    unsigned short int high_count;
    unsigned short int scale;
} SYMBOL;

void build_model();
void scale_counts();
void build_totals();
void count_bytes();
void output_counts();
void input_counts();
void convert_int_to_symbol();
void get_symbol_scale();
void initialize_arithmetic_encoder();
void encode_symbol();
void flush_arithmetic_encoder();
void initialize_arithmetic_decoder();
void remove_symbol_from_stream();
```

→

ลิสติงที่ 3 (ต่อ)

```
int convert_symbol_to_int();
short int get_current_count();

#define END_OF_STREAM 256
short int totals[ 258 ];            /* The cumulative totals */

char *CompressionName = "Arithmetic coding with order 0 model.";
char *Usage = "infile outfile [-c/-e]\n\t-c Compression\n\t-e Expansion\n";
/*
 * Compress file by arithmatic coding.
 */
void CompressFile( input, output, argc, argv )
FILE *input;
BIT_FILE *output;
int argc;
char *argv[];
{
    int c;
    SYMBOL s;

    build_model( input, output->file );
    initialize_arithmetic_encoder();

    while ( ( c = getc( input ) ) != EOF ) {
      convert_int_to_symbol( c, &s );
      encode_symbol( output, &s );
    }
    convert_int_to_symbol( END_OF_STREAM, &s );
    encode_symbol( output, &s );
    flush_arithmetic_encoder( output );
    OutputBits( output, 0L, 16 );
}

/*
 * Expand file by arithmatic coding.
 */
void ExpandFile( input, output, argc, argv )
BIT_FILE *input;
FILE *output;
int argc;
char *argv[];
{
    SYMBOL s;
    int c;
    int count;

    input_counts( input->file );
    initialize_arithmetic_decoder( input );
    for ( ; ; ) {
      get_symbol_scale( &s );
      count = get_current_count( &s );
      c = convert_symbol_to_int( count, &s );
      if ( c == END_OF_STREAM )
            break;
      remove_symbol_from_stream( input, &s );
        putc( (char) c, output );
    }
}

/*
 * This is the routine that is called to scan the input file, scale the
 * counts, build the totals array, the output the scaled counts to the output file.
 */
void build_model( input, output )
FILE *input;
FILE *output;
{
    unsigned long counts[ 256 ];
    unsigned char scaled_counts[ 256 ];

    count_bytes( input, counts );
    scale_counts( counts, scaled_counts );
    output_counts( output, scaled_counts );
    build_totals( scaled_counts );
}
```

→

ลิสติงที่ 3 (ต่อ)

```
/*
 * This routine runs through the file and counts the appearances of each character.
 */
#ifndef SEEK_SET
#define SEEK_SET 0
#endif
void count_bytes( input, counts )
FILE *input;
unsigned long counts[];
{
    long input_marker;
    int i;
    int c;

    for ( i = 0 ; i < 256 ; i++ )
        counts[ i ] = 0;
    input_marker = ftell( input );
    while ( ( c = getc( input )) != EOF )
    , counts[ c ]++;
    fseek( input, input_marker, SEEK_SET );
}

/*
 * This routine is called to scale the counts down.
 */
void scale_counts( counts, scaled_counts )
unsigned long counts[];
unsigned char scaled_counts[];
{
    int i;
    unsigned long max_count;
    unsigned int total;
    unsigned long scale;

    max_count = 0;
    for ( i = 0 ; i < 256 ; i++ )
       if ( counts[ i ] > max_count )
        max_count = counts[ i ];
    scale = max_count / 256;
    scale = scale + 1;
    for ( i = 0 ; i < 256 ; i++ ) {
        scaled_counts[ i ] = (unsigned char ) ( counts[ i ] / scale );
        if ( scaled_counts[ i ] == 0 && counts[ i ] != 0 )
            scaled_counts[ i ] = 1;
    }

    total = 1;
    for ( i = 0 ; i < 256 ; i++ )
        total += scaled_counts[ i ];
    if ( total > ( 32767 - 256 ) )
        scale = 4;
    else if ( total > 16383 )
        scale = 2;
    else
        return;
    for ( i = 0 ; i < 256 ; i++ )
        scaled_counts[ i ] /= scale;
}

/*
 * The counts for the characters in the file are in the counts array, and we
 * know that there will be a single instance of the EOF symbol.
 */
void build_totals( scaled_counts )
unsigned char scaled_counts[];
{
    int i;

    totals[ 0 ] = 0;
    for ( i = 0 ; i < END_OF_STREAM ; i++ )
        totals[ i + 1 ] = totals[ i ] + scaled_counts[ i ];
    totals[ END_OF_STREAM + 1 ] = totals[ END_OF_STREAM ] + 1;
}
```

→

ลิสติงที่ 3 (ต่อ)

```
/*
 * In order for the compressor to build the same model, I have to store the symbol
 * counts in the compressed file so the expander can read them in. In order to
 * save space, I don't save all 256 symbols unconditionally. The format used
 * to store counts looks like this:
 * start, stop, counts, start, stop, counts, ... 0
 */
void output_counts( output, scaled_counts )
FILE *output;
unsigned char scaled_counts[];
{
    int first;
    int last;
    int next;
    int i;

    first = 0;
    while ( first < 255 && scaled_counts[ first ] == 0 )
        first++;

    for ( ; first < 256 ; first = next ) {
      last = first + 1;
      for ( ; ; ) {
          for ( ; last < 256 ; last++ )
              if ( scaled_counts[ last ] == 0 )
              break;
          last--;
          for ( next = last + 1; next < 256 ; next++ )
              if ( scaled_counts[ next ] != 0 )
              break;
          if ( next > 255 )
            break;
          if ( ( next - last ) > 3 )
            break;
          last = next;
      };
        if ( putc( first, output ) != first )
          fatal_error( "Error writing byte counts\n" );
      if ( putc( last, output ) != last )
          fatal_error( "Error writing byte counts\n" );
      for ( i = first ; i <= last ; i++ ) {
          if ( putc( scaled_counts[ i ], output ) !=
              (int) scaled_counts[ i ] )
            fatal_error( "Error writing byte counts\n" );
      }
    }
    if ( putc( 0, output ) != 0 )
        fatal_error( "Error writing byte counts\n" );
}

/*
 * When expanding, I have to read in the same set of counts. This is quite a bit easier
 * that the process of writing them out, since no decision making needs to be done.
 */
void input_counts( input )
FILE *input;
{
    int first;
    int last;
    int i;
    int c;
    unsigned char scaled_counts[ 256 ];

    for ( i = 0 ; i < 256 ; i++ )
        scaled_counts[ i ] = 0;
    if ( ( first = getc( input ) ) == EOF )
      fatal_error( "Error reading byte counts\n" );
    if ( ( last = getc( input ) ) == EOF )
      fatal_error( "Error reading byte counts\n" );
    for ( ; ; ) {
      for ( i = first ; i <= last ; i++ )
          if ( ( c = getc( input ) ) == EOF )
          fatal_error( "Error reading byte counts\n" );
        else
```

→

ลิสติงที่ 3 (ต่อ)

```
                scaled_counts[ i ] = (unsigned char) c;
            if ( ( first = getc( input ) ) == EOF )
                fatal_error( "Error reading byte counts\n" );
            if ( first == 0 )
                break;
            if ( ( last = getc( input ) ) == EOF )
                fatal_error( "Error reading byte counts\n" );
        }
        build_totals( scaled_counts );
}

/*
 * These four variables define the current state of the arithmetic coder/decoder.
 * They are assumed to be 16 bits long. Note that by declaring them as short ints,
 * they will actually be 16 bits on most 80X86.
 */
static unsigned short int code;
static unsigned short int low;
static unsigned short int high;
long underflow_bits;

/*
 * This routine must be called to initialize the encoding process.
 */
void initialize_arithmetic_encoder()
{
    low = 0;
    high = 0xffff;
    underflow_bits = 0;
}

/*
 * At the end of the encoding process, there are still significant bits left
 * in the high and low registers.
 */
void flush_arithmetic_encoder( stream )
BIT_FILE *stream;
{
    OutputBit( stream, low & 0x4000 );
    underflow_bits++;
    while ( underflow_bits-- > 0 )
        OutputBit( stream, ~low & 0x4000 );
}

/*
 * Finding the low count, high count, and scale for a symbol is really easy,
 * because of the way the totals are stored.
 */
void convert_int_to_symbol( c, s )
int c;
SYMBOL *s;
{
    s->scale = totals[ END_OF_STREAM + 1 ];
    s->low_count = totals[ c ];
    s->high_count = totals[ c + 1 ];
}

void get_symbol_scale( s )
SYMBOL *s;
{
    s->scale = totals[ END_OF_STREAM + 1 ];
}

/*
 * During decompression, we have to search through the table until we find the
 * symbol that straddles the "count" parameter. When it is found, it is returned.
 */
int convert_symbol_to_int( count, s)
int count;
SYMBOL *s;
{
    int c;
```

→

ฉบับที่ 3 (ต่อ)

```
    for ( c = END_OF_STREAM ; count < totals[ c ] ; c-- )
        ;
    s->high_count = totals[ c + 1 ];
    s->low_count = totals[ c ];
    return( c );
}

/*
 * This routine is called to encode a symbol.
 */
void encode_symbol( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;
    range = (long) ( high-low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );
    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
            OutputBit( stream, high & 0x8000 );
            while ( underflow_bits > 0 ) {
                OutputBit( stream, ~high & 0x8000 );
                underflow_bits--;
            }
        }
        else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
            underflow_bits += 1;
            low &= 0x3fff;
            high |= 0x4000;
        } else
            return ;
        low <<= 1;
        high <<= 1;
        high |= 1;
    }
}

/*
 * When decoding, this routine is called to figure out which symbol is presently waiting
 * to be decoded. This routine expects to get the current model scale in the s->scale
 * parameter, and it returns a count that corresponds to the present floating point code:
 */
short int get_current_count( s )
SYMBOL *s;
{
    long range;
    short int count;

    range = (long) ( high - low ) + 1;
    count = (short int)
            ((((long) ( code - low ) + 1 ) * s->scale-1 ) / range );
    return( count );
}

/*
 * This routine is called to initialize the state of the arithmetic decoder.
 */
void initialize_arithmetic_decoder( stream )
BIT_FILE *stream;
{
    int i;

    code = 0;
    for ( i = 0 ; i < 16 ; i++ ) {
        code <<= 1;
        code += InputBit( stream );
    }
    low = 0;
    high = 0xffff;
}
```

→

ลิสติงที่ 3 (ต่อ)

```
/*
 * Just figuring out what the present symbol is doesn't remove it from the input
 * bit stream. After the character has been decoded, this routine has to be
 * called to remove it from the input stream.
 */
void remove_symbol_from_stream( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;

    range = (long)( high - low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );
    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
        }
        else if ((low & 0x4000) == 0x4000  && (high & 0x4000) == 0 ) {
            code ^= 0x4000;
            low   &= 0x3fff;
            high  |= 0x4000;
        } else
            return;
        low <<= 1;
        high <<= 1;
        high |= 1;
        code <<= 1;
        code += InputBit( stream );
    }
}
```

วางแผงแล้ววันนี้
นิตยสารรายเดือน
Internet Internet Internet
เข้มด้วยเนื้อหาและสาระ บทความที่โดดเด่น
บริษัท ซีเอ็ดยูเคชั่น (มหาชน) จำกัด
รับประกันคุณภาพ